Class Project 1- Graph Analytics

Introduction to Big Data & Analytics
CSCI 6444

Group 13- Amish Raj, Arham Ashfaq
Submitted to Professor Stephen Kaisler

1. Dataset Description

The dataset is a graph of links of opinions from the SOC-E website. In this dataset, the names have been removed and replaced by numbers. It is a medium sized dataset where each row has the format <node1>, <node2>, #edges and there are approximately 10 million nodes. Since for this dataset, #edges is 1 for each row, we will be removing the #edges column after loading the dataset as a data frame.

2. Install the igraph package from one of the CRAN mirrors. Determine how to create a graph and plot. Show the plot in your report.

We first install **igraph** package from one of the CRAN mirrors as described in the problem statement. We can do so by typing **"install.packages(igraph)"** in the console and then **"library(igraph)"** to use it in our project. We get an output of the form below-

```
> install.packages("igraph")
trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-arm64/contrib/4.2/igraph_1.4.1.
tgz'
Content type 'application/x-gzip' length 8231069 bytes (7.8 MB)
==================================================
downloaded 7.8 MB


The downloaded binary packages are in
        /var/folders/d5/w02vn_rj1hv2m765h0gj7qdc0000gn/T//Rtmpzc6o8N/downloaded_package
s
```

```
> library(igraph)

Attaching package: 'igraph'

The following objects are masked from 'package:stats':

    decompose, spectrum

The following object is masked from 'package:base':

    union
```

Next, we **load the dataset** (keeping the **soc-Epinions1_adj.tsv** file in the same working directory), **convert it to a matrix**, and **extract two vectors** v1 and v2. By doing this, we get the vertices in the form of vectors. We do not use the third column since that denotes the #edges which is 1 as we know for all rows.

Once we have the vectors, we can create a data frame and then a graph from it using igraph's **graph_from_data_frame()** function. In this manner, we successfully create our graph.

The code snippet for these steps is given below-

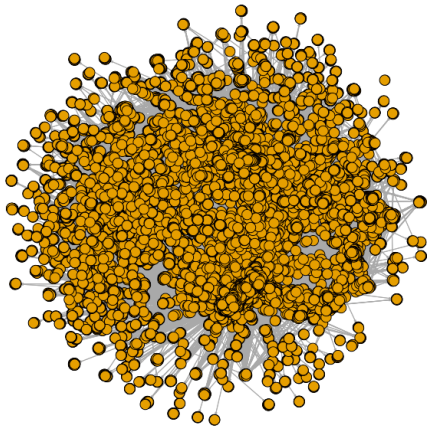
```
> opinions <- read.table("soc-Epinions1_adj.tsv")
> optab <- as.matrix(opinions)
> v1 <- optab[,1]
> v2 <- optab[,2]
> v3 <- optab[,3]
> relations <- data.frame(from=v1, to=v2)
> g <- graph_from_data_frame(relations, directed= TRUE)


> plot(g, vertex.size=5, edge.arrow.size=0.1)
```

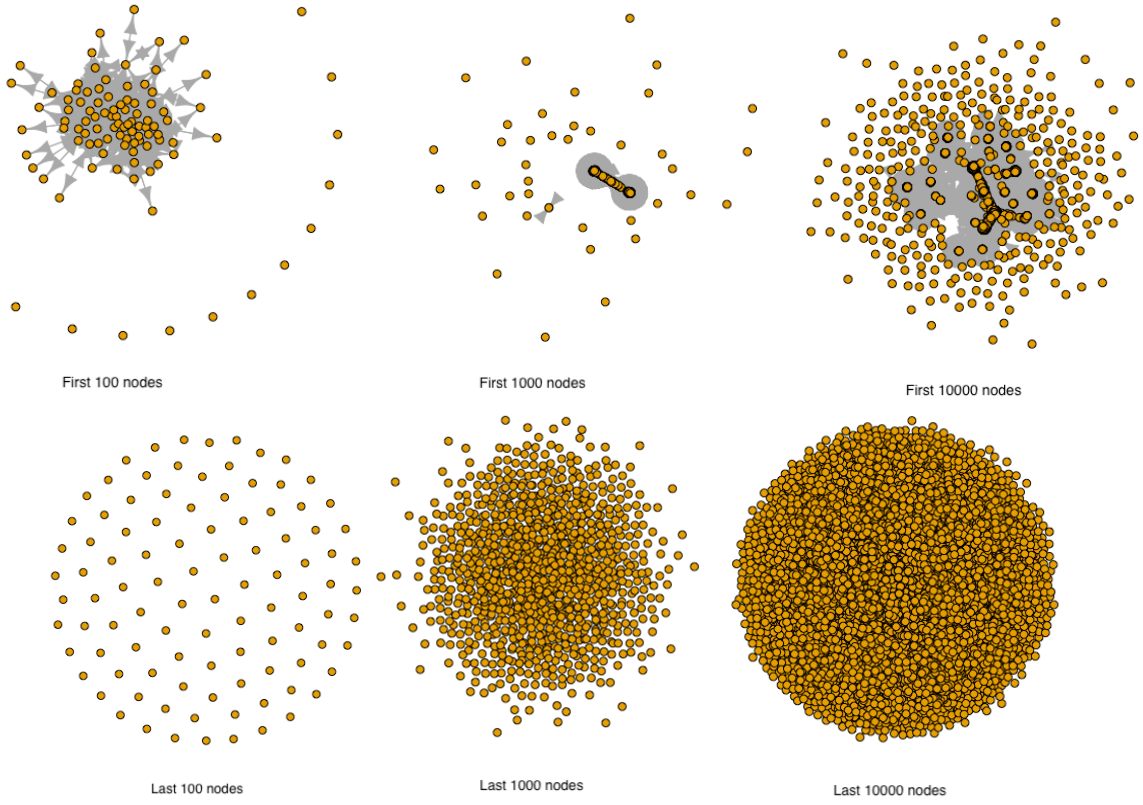
```
> g
IGRAPH dc96229 DN-- 75879 811480 --
+ attr: name (v/c)
+ edges from dc96229 (vertex names):
 [1] 3   ->1 4   ->1 115 ->1 150 ->1 182 ->1 226 ->1 282 ->1 337 ->1 371 ->1
[10] 448 ->1 559 ->1 670 ->1 780 ->1 826 ->1 875 ->1 891 ->1 897 ->1 925 ->1
[19] 1002->1 1111->1 1112->1 1122->1 1223->1 1334->1 1445->1 1556->1 1667->1
[28] 1778->1 1834->1 1889->1 2000->1 2001->1 2080->1 2111->1 2149->1 2222->1
[37] 2223->1 2242->1 2334->1 2346->1 2434->1 2445->1 2501->1 2534->1 2556->1
[46] 2601->1 2623->1 2667->1 2727->1 2778->1 2811->1 2889->1 3000->1 3041->1
[55] 3089->1 3110->1 3111->1 3145->1 3222->1 3305->1 3333->1 3334->1 3379->1
[64] 3410->1 3445->1 3534->1 3556->1 3574->1 3667->1 3778->1 3889->1 3989->1
+ ... omitted several edges
```

As we can see in the code snippet above, once our **graph "g"** is ready, we proceed to plot the graph. We will be using the notation "g" for our graph later on as well.

Initially, the generated plot from simply running plot(g) had a very high execution time, and the resulting plot was extremely convoluted. Therefore, we reduce vertex size to 5 and edge arrow size to 0.1. This gives us a much more comprehensible graph as shown below.



However, in order to understand the plot further and simplify the graph representation, we compiled the plots of the first 100, 1000, 10000, and the last 100, 1000 and 10000 nodes. These are represented as follows-



3. Apply the functions shown in the Introduction to Graph Analytics document on Blackboard on the graph generated from the data set.

Following the rubric, we apply the functions in the Introduction to Graph Analytics document and describe the results as screenshots, along with a description what the results can tell us about the problem domain.

Some of the outputs are too large and hence, most rows get omitted during printout or in the screenshot.

a. V(g): Vertices of a Graph

```
> V(g)
+ 75879/75879 vertices, named, from dc96229:
   [1] 3     4     115   150   182   226   282   337   371   448   559   670
  [13] 780   826   875   891   897   925   1002  1111  1112  1122  1223  1334
  [25] 1445  1556  1667  1778  1834  1889  2000  2001  2080  2111  2149  2222
  [37] 2223  2242  2334  2346  2434  2445  2501  2534  2556  2601  2623  2667
  [49] 2727  2778  2811  2889  3000  3041  3089  3110  3111  3145  3222  3305
  [61] 3333  3334  3379  3410  3445  3534  3556  3574  3667  3778  3889  3989
  [73] 4000  4111  4158  4222  4333  4341  4444  4445  4556  4563  4667  4778
  [85] 4889  4978  5000  5111  5222  5289  5333  5367  5385  5444  5456  5489
  [97] 5554  5555  5556  5633  5666  5667  5744  5766  5777  5778  5804  5888
 [109] 5911  5922  5999  6000  6110  6155  6221  6244  6266  6332  6346  6355
+ ... omitted several vertices
```

The **V(g)** function returns the vertices of a graph, and we see that in graph g we have 75879 vertices. Since names have been removed and replaced by numbers in this dataset, that is what we see as labels for the nodes here.

b. E(g): Edges of a Graph

```
> E(g)
+ 811480/811480 edges from dc96229 (vertex names):
 [1] 3   ->1 4   ->1 115 ->1 150 ->1 182 ->1 226 ->1 282 ->1 337 ->1 371 ->1
[10] 448 ->1 559 ->1 670 ->1 780 ->1 826 ->1 875 ->1 891 ->1 897 ->1 925 ->1
[19] 1002->1 1111->1 1112->1 1122->1 1223->1 1334->1 1445->1 1556->1 1667->1
[28] 1778->1 1834->1 1889->1 2000->1 2001->1 2080->1 2111->1 2149->1 2222->1
[37] 2223->1 2242->1 2334->1 2346->1 2434->1 2445->1 2501->1 2534->1 2556->1
[46] 2601->1 2623->1 2667->1 2727->1 2778->1 2811->1 2889->1 3000->1 3041->1
[55] 3089->1 3110->1 3111->1 3145->1 3222->1 3305->1 3333->1 3334->1 3379->1
[64] 3410->1 3445->1 3534->1 3556->1 3574->1 3667->1 3778->1 3889->1 3989->1
[73] 4000->1 4111->1 4158->1 4222->1 4333->1 4341->1 4444->1 4445->1 4556->1
[82] 4563->1 4667->1 4778->1 4889->1 4978->1 5000->1 5111->1 5222->1 5289->1
+ ... omitted several edges
```

The **E(g)** function is like V(g) except that it returns the edges of a graph, and in our graph g we have 811480 edges. This gives us more information about the dataset.

c. get.adjacency()

```
> g.adj= igraph::get.adjacency(g)
> g.adj
75879 x 75879 sparse Matrix of class "dgCMatrix"
   [[ suppressing 32 column names '3', '4', '115' ... ]]
   [[ suppressing 32 column names '3', '4', '115' ... ]]

3   . 1 . . 1 1 . 1 . 1 . . . . . . . . . 1 1 . . . . . . . . 1 . 1 ......
4   1 . . . . 1 . 1 . 1 . 1 1 . . 1 . . 1 1 . . . . . . . . 1 . 1 . . ......
115 . . . . . . . 1 . 1 1 . . . . 1 . . . 1 . . . . 1 . . . . 1 . . ......
150 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1 . ......
182 1 . . . . . . 1 1 1 . . . . . 1 . . . . . . 1 1 1 . . . . . . . ......
226 1 1 . . . . . 1 1 1 . . 1 . . 1 . . 1 1 . . . . . . . 1 . 1 . 1 ......
282 . . . . . . . . . . . . . . . . . . . . . 1 . . . . . . . . . . ......
337 1 1 1 . 1 1 . . . 1 . . 1 . . 1 . . 1 1 1 . . . . . . 1 1 1 . . ......
371 . . . . 1 1 . . . . . . . . . . . . 1 . 1 . . . . . . . . . . . ......
448 1 1 1 . 1 1 . 1 . . . . 1 . . . . . . . . 1 . . . . . . 1 1 1 . . ......
559 . . 1 . . . . . . . . . . . . . . . . . 1 . 1 . . . . 1 . . . . . ......
670 . 1 . . . . . . . . . . . . . . . . . . . 1 . . . . . . . 1 . . . ......
780 . 1 . . . 1 . 1 . 1 . . . . . . . . . 1 . . . . . 1 . . . . . . ......
826 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
875 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
891 . 1 1 . 1 1 . 1 . . . . . . . . . . . . 1 . . . . 1 . . 1 . 1 1 . ......

 ..............................
 ........suppressing 75847 columns and 75848 rows in show(); maybe adjust 'options(
 ..............................
   [[ suppressing 32 column names '3', '4', '115' ... ]]

7405  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
7409  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
7411
```

The **get.adjacency()** function returns the adjacency matrix for a graph. Here we get our graph g's 758789 x 75879 adjacency matrix. The adjacency matrix gives us an idea about the density of our graph and even helps us identify any isolated vertices.
For example, we can notice that the node 875 is relatively less well connected to other nodes as opposed to node 337. In this manner, we can use this information to manipulate the graph for further analysis.

d. gden()

```
> g.density= gden(relations)
> g.density
[1] 0.1569413
```

The **gden()** function computes the density of a graph, considering and adjusting for the type of graph. Here, we can see the density of our graph is 0.1569413, which means that our graph is relatively sparse i.e., the number of edges that exist are very little compared to all the edges that could exist.

e. edge_density()

```
> igraph:: edge_density(g)
[1] 0.000140942
> igraph:: edge_density(g, loops=T)
[1] 0.0001409401
```

The density of a graph is defined as the ratio of the number of edges in the graph to the maximum number of edges that could exist in the graph, which is what the **edge_density()** function computes. We can see that even by allowing loops in the graph,

the density is almost the same, which is an indicator that this graph probably does not contain loops (and therefore, might be simple graph).

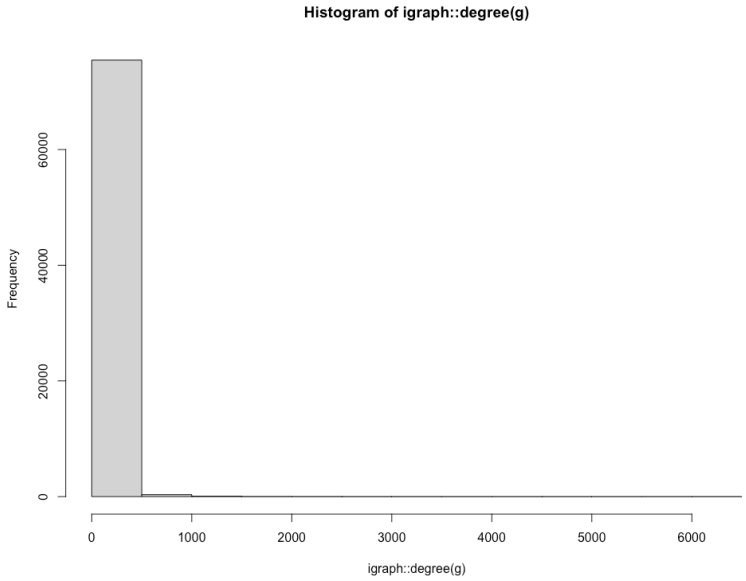
f. degree()

```
> igraph::degree(g)
    3     4   115   150   182   226   282   337   371   448   559
  572   690   686    48   636   880   266   924   342   826   618
  925  1002  1111  1112  1122  1223  1334  1445  1556  1667  1778
  372   332   448   498    22   362   830   508    82   366   432
 2149  2222  2223  2242  2334  2346  2434  2445  2501  2534  2556
   26   834   542    30    24   176   252   422  1046   224   602
 2889  3000  3041  3089  3110  3111  3145  3222  3305  3333  3334
  396   730    36   314    36  1736   814   104    20   732   820
 3667  3778  3889  3989  4000  4111  4158  4222  4333  4341  4444
  390   262  1296    96  2410   236     8   664   364    20   412
 4978  5000  5111  5222  5289  5333  5367  5385  5444  5456  5489
  362  1328    86    80   760    36   602    66    52    60   588
 5744  5766  5777  5778  5804  5888  5911  5922  5999  6000  6110
  352   370     8   432    24    56   522   764   612   650   288
 6355  6443  6554  6665  6666  6711  6777  6789  6855  6888  6922
```

The **degree()** function is defined in several packages however here, we use the one defined in the igraph package. It returns the number of adjacent edges for each vertex.

From the results, we get information about the general structure and density of the graph, which we can infer by looking at the degrees of the different vertices. Let us visualize the degrees of the nodes of the graph using a histogram.

```
> hist(igraph::degree(g))
```



From this we can infer that most nodes have a degree between 0 and 500, and the rest have the degrees between 500-1000.

g. centr_betw() : Betweenness Centrality

```
> g.between= igraph:: centr_betw(g)
> g.between
$res
   [1] 3.151719e+06 3.981881e+06 3.266552e+06 1.206334e+05 7.
   [8] 8.122346e+06 2.725038e+06 5.273894e+06 3.744818e+06 4.
  [15] 8.783339e+02 1.474652e+06 3.244700e+05 3.461340e+06 2.
  [22] 9.004451e+05 2.007443e+06 1.553106e+07 2.858003e+06 1.
  [29] 2.053627e+06 2.216830e+06 5.637427e+07 2.877706e+06 1.
  [36] 5.445164e+06 4.715068e+06 4.811867e+05 1.129239e+04 5.
  [43] 2.380167e+07 1.479723e+06 2.850635e+06 4.453057e+06 9.
  [50] 2.534605e+07 3.859493e+04 2.311553e+06 6.605852e+06 7.
  [57] 4.907386e+07 1.259179e+07 3.950718e+05 2.408225e+03 1.
  [64] 1.527451e+05 1.285051e+06 1.412434e+06 2.400280e+06 1.
  [71] 2.173360e+07 3.344439e+05 8.976368e+07 1.215126e+06 1.
 [ reached getOption("max.print") -- omitted 74879 entries ]

$centralization
[1] 0.0721838

$theoretical_max
[1] 4.368596e+14
```

The **betweenness centrality** tells us about the number of shortest paths that pass through each vertex. In general, we can see that our graph g has relatively lower betweenness for its nodes, which means that there are fewer nodes with high betweenness (nodes that have a strong influence on the connectivity of the network).
This is desirable in some scenarios, however in other scenarios it might also mean that the nodes in the graph are more isolated from the rest – which might not be desirable.

h. centr_clo(): Closeness Centrality

```
> g.closeness = igraph::centr_clo(g)
> g.closeness
$res
   [1] 0.3235830 0.3182144 0.3204927 0.2823251 0.3207840 0.3293043 0.3087729 0.32294
  [10] 0.3246296 0.3207826 0.2975331 0.3315679 0.2943417 0.2642594 0.3140524 0.27166
  [19] 0.3123369 0.3177786 0.3239823 0.2667126 0.3209699 0.3424100 0.3161158 0.29236
  [28] 0.3178638 0.3058456 0.3158092 0.3518479 0.3082924 0.2727371 0.3334139 0.26993
  [37] 0.3230250 0.2749230 0.2786445 0.3031511 0.3050439 0.3111585 0.3431999 0.31619
  [46] 0.3198981 0.3099168 0.3268967 0.2949711 0.3353028 0.2754350 0.3199953 0.32843
  [55] 0.3058234 0.2739839 0.3554810 0.3386626 0.2808288 0.2706291 0.3264200 0.33649
  [64] 0.2685268 0.3009913 0.3027798 0.3189461 0.2903622 0.3216570 0.3056891 0.33889
  [73] 0.3585687 0.3120068 0.2611470 0.3222253 0.3155636 0.2713161 0.3162277 0.33030
  [82] 0.2834219 0.3070088 0.3623253 0.2915404 0.3218726 0.3302819 0.3036437 0.30088
  [91] 0.2756131 0.3384390 0.2847589 0.2789292 0.2934538 0.3306302 0.3060861 0.27792
 [100] 0.3219778 0.2691621 0.3270278 0.3197081 0.3177414 0.2569664 0.3195344 0.27535
 [109] 0.3292143 0.3159775 0.3212213 0.3202370 0.3009663 0.3116672 0.3051727 0.31850
 [118] 0.2670825 0.2597710 0.3247977 0.2874810 0.3021756 0.3127321 0.2596661 0.29426
 [127] 0.3016686 0.3420195 0.2708116 0.3397499 0.2922264 0.2856067 0.3222595 0.31313
 [136] 0.3136331 0.3109749 0.2923367 0.3370020 0.3063927 0.2561172 0.3188201 0.26337
 [145] 0.3235098 0.2732803 0.2679890 0.3040379 0.2842011 0.2759810 0.2950433 0.25954
 [ reached getOption("max.print") -- omitted 74879 entries ]
$centralization
[1] 0.7633521

$theoretical_max
[1] 75877
```

The **centr_clo()** function returns the closeness centrality of the various nodes of the graph.. The higher the closeness, the closer the node is to other nodes in the graph.
Here, we observe an average closeness centrality of 0.3, which indicates that the nodes in the graph g are fairly well connected and can reached from most other nodes. We can visually see the proof of this as well from our plots earlier.

i. shortest.paths: Shortest Path Between Two Nodes

```
g.sp= igraph::shortest.paths(g)
g.sp
```

```
     41562 41570 41571 41574 41578 41580 41583 41588 41589 41590 41591 41596 41597 41599 41600 41601
     41603 41604 41605 41608 41610 7120 7123 7125 7126 72151 41611 41613 41614 41616 7134 7138 7141
     7143 7145 41621 41623 41624 41628 41629 41633 41634 41635 41636 7146 41639 41640 41641 7150 7151
     7152 7154 7156 41643 41644 41645 51309 51320 51365 51387 51531 7161 7162 7163 7165 7167 7169
     41650 41651 41652 41654 41655 41657 41658 41659 41662 41667 41668 41669 41671 41675 7178 7179
     7180 7181 7182 7184 7187 7194 41677 41679 41680 41681 41682 41683 41684 41685 41688 41689 7200
     7202 41691 41693 41694 41695 41700 41701 41704 7204 7208 7214 7219 41707 41708 41710 41712 41713
     41714 41715 7232 41719 41721 41722 51609 51631 51720 51731 7235 7238 41723 41724 41725 7259 41726
     7261 7264 7267 7269 7278 7279 70684 70685 70686 7280 7281 7284 7287 7291 7292 41728 41729 41732
     7294 7297 41733 41735 41736 7494 7495 41743 41744 41745 41746 41747 41748 7302 7303 7304 7331
     7345 70873 41750 41752 41754 41756 41757 41759 41760 41761 41762 7348 7350 7351 7356 7357 7358
     7359 7360 7361 7362 7363 7364 7370 41763 41766 41767 41768 41769 41770 41772 7373 7374 7375 7376
     7378 7387 7389 7397 7400 7403 7405 7409 7411 41774 41777 41780 41781 41782 41783 41784 41785
     41786 41789 41790 41792
[ reached getOption("max.print") -- omitted 75879 rows ]
```

The function igraph::shortest.paths(g) provides us the shortest path between any two nodes in a graph g. Here they are represented as a matrix for a shortest path from node i to node j.
Using this, we can locate vertices that are more central than others or find the more isolated nodes. For example, in our shortest path output above, the last path with value 41792 is less preferable than a shortest path of 7400 in the row right above it. There are some entries with over 60000 as the shortest path, and based on the use case, these observations are very useful.

j. get.shortest.path(): Get shortest paths between vertices in a graph

```
$vpath[[998]]
+ 3/75879 vertices, named, from 859a294:
[1] 182   12187 32442

$vpath[[999]]
+ 3/75879 vertices, named, from 859a294:
[1] 182   1334  32487

$vpath[[1000]]
+ 3/75879 vertices, named, from 859a294:
[1] 182   448   32553

[ reached getOption("max.print") -- omitted 74879 entries ]
```

```
> igraph::get.shortest.paths(g, 5)
```

The shortest paths function here helps us get the length as well as the actual shortest paths from node 5 in our graph g. This is very useful as we can gauge how well the node (5, in this case) is connected to other vertices. Here the node 5 has the shortest path length value of 3 or 4 for most vertices in the output, indicating it is relatively well connected.

k. max_cliques()

```
> node <- c(20)
> g.adj_graph <- igraph::graph_from_adjacency_matrix(g.adj)
> g.20clique= igraph::max_cliques(g.adj_graph, min=NULL, max=NULL, subset=node)
Warning message:
In igraph::max_cliques(g.adj_graph, min = NULL, max = NULL, subset = node) :
  At core/cliques/maximal_cliques_template.h:269 : Edge directions are ignored for maximal clique calculation.
> g.20clique
[[1]]
+ 2/75879 vertices, named, from 010c0ef:
[1] 43492 1310
```

The max_cliques() function from the igraph package helps us identify the largest cliques in a graph, along with the size of the largest clique. Here, for node 20 the largest clique is 2 with the nodes 43492 and 1310. A clique of 2 tells us that these nodes are highly

connected to each other but not necessarily to the rest of the graph. A higher clique would indicate densely connected nodes.

l. clique_num(): Largest clique in the graph

```
> g.lgcliques = igraph::clique_num(g.adj_graph)
Warning message:
In igraph::clique_num(g.adj_graph) :
  At core/cliques/maximal_cliques_template.h:269 : Edge directions are ignored for maximal clique calculation.
> g.lgcliques
[1] 23
```

The clique_num() function calculates the size of the largest clique in the graph, which in the case of our graph, is 23.

m. Simplify() and is.simple()

```
> is.simple(g)
[1] TRUE
> sg <- simplify(g)
> is.simple(sg)
[1] TRUE
```

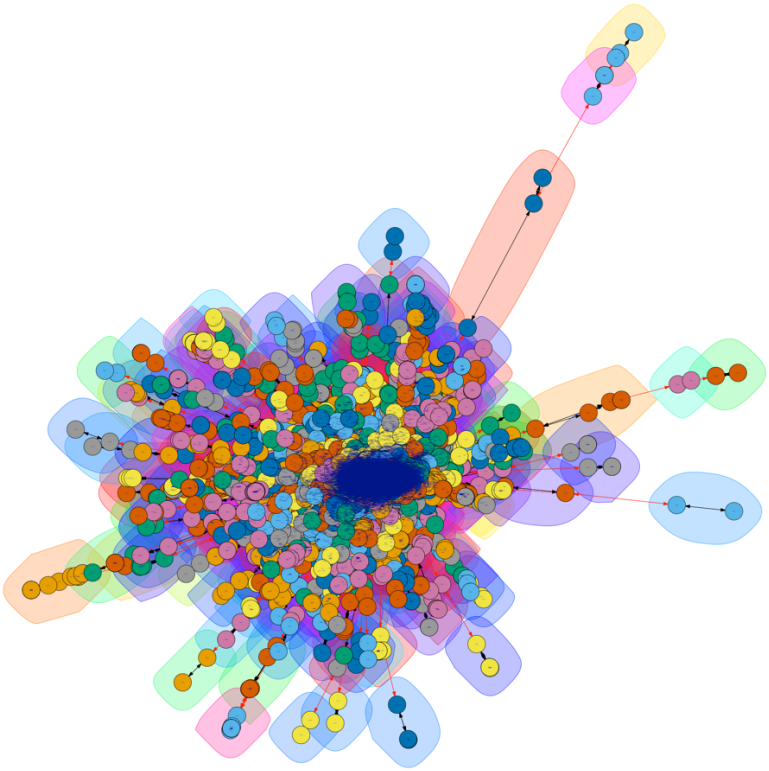
```
> is.simple(g)
[1] TRUE
```

The is.simple() method tells us if our graph is simple i.e., if our graph has no loops and multiple edges between vertices. Without performing any simplification, we notice that our graph is simple. This is verified when we apply the simplify() function on the graph after which, the is.simple() method still returns true.

n. Detecting Structures using walktrap.community()

```
wc<-walktrap.community(g)

plot(wc, g, vertex.size=5, vertex.label.cex=0.2, edge.arrow.size=0.1, layout=layout.fruchterman.reingold)
```



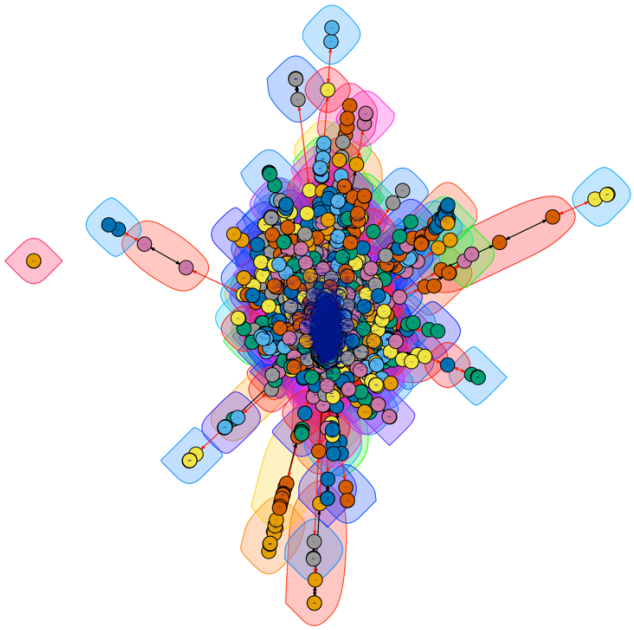
The **walktrap.community()** function tries to find densely connected subgraphs or communities in a graph through random walks since short random walks tend to stay in the same community.
In the plot above, we can see the various communities highlighted. Most of the subgraphs are dense and close together towards the center, whereas a few outliers exist outside the blob.

4. Graph Simplification

The first step we took in order to simplify our graph was to apply the simplify() function. However, as we demonstrated before, this does not give us a productive result as our graph is already simple i.e. it has no loops and multiple edges between vertices.

```
> is.simple(g)
[1] TRUE
> sg <- simplify(g)
> is.simple(sg)
[1] TRUE
```

This is why, if we try to find communities in the simplified graph and plot it, we don't see a lot of difference from the previous plot of communities.

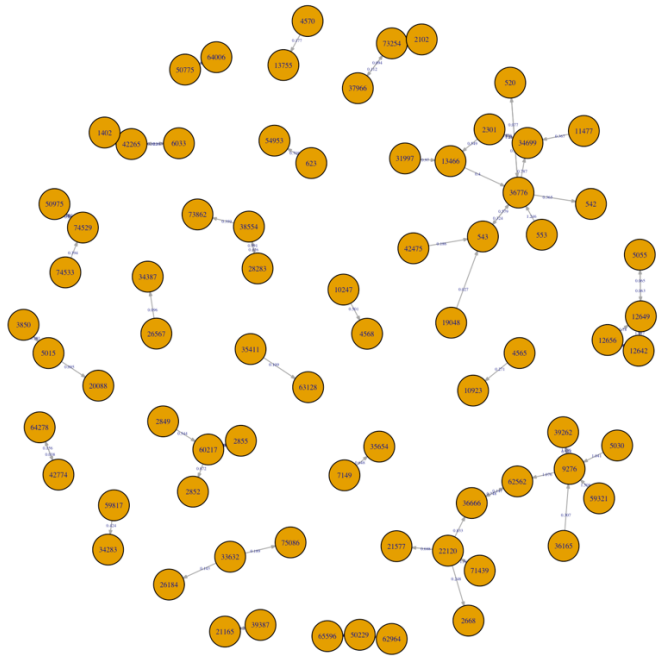


So to go a step further, we thought to remove empty entries from the matrix i.e. remove isolated nodes with degree 0. However, we notice that the graph has no nodes with degree 0.

```
> sum(degree(g)==0)
[1] 0
```

In order to perform some sort of real-world analysis on the graph, we can randomly assign weights to the vertices and the edges (as shown in the rubric on the astrocollab dataset). Upon doing that, we can fetch get a subgraph with only vertices having weight greater than 2.2 with the help of **induced.subgraph()** function.

```
 > g_duplicate <- g
> E(g_duplicate)$weight <- rnorm(ecount(g_duplicate))
> V(g_duplicate)$weight <- rnorm(vcount(g_duplicate))
> sg_duplicate <- induced.subgraph(g_duplicate, which(V(g_duplicate)$weight > 2.2))
> plot(delete.vertices(sg_duplicate, degree(sg_duplicate)==0), edge.label = round(E(sg_duplicate)$weight,
3), edge.arrow.size=0.1, vertex.label.cex=0.5, edge.label.cex=0.3)
```

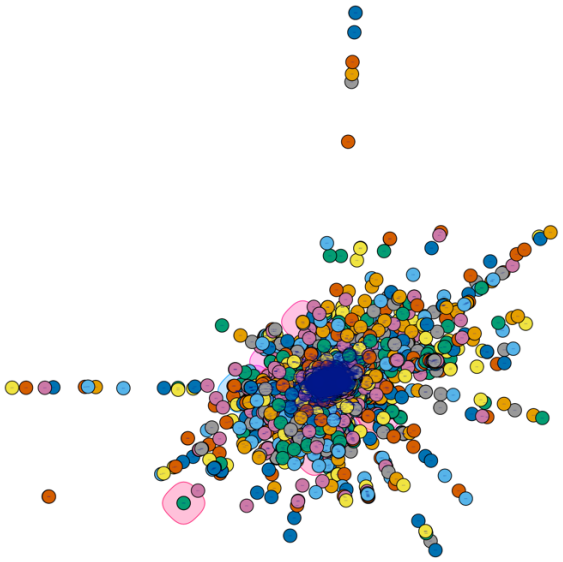


This subgraph plot visualizes some of the highly weighted vertices in the network i.e., vertices that are more significant in the network.

Note that here, we have duplicated the graph g (**g_duplicate**) and taken a subgraph of it (**sg_duplicate**) separate from graph g for convenience. We also remove the negative edge weights from sg_duplicate.

When we try to find communities in the subgraph sg_duplicate via random walks, we receive the following result-

```
> wc_sg_duplicate<-walktrap.community(sg_duplicate)
> plot(wc_sg_duplicate, g, vertex.size=5, vertex.label.cex=0.1, edge.arrow.size=0.1, layout=layout.fruchterm
an.reingold)
```



5. Determining the alpha centrality, central node in the graph, longest paths, largest cliques, egos, and power centrality

a) Alpha Centrality:

We try to find the **alpha centrality** of our subgraph **sg_duplicate** using igraph's **alpha_centrality()** function. The alpha centrality on our graph **g_duplicate** was computationally too expensive and did not yield a result even after running for hours. We see the following result for the sg_duplicate alpha centrality:

```
> acsg_duplicate <- alpha_centrality(sg_duplicate)
> sort(acsg_duplicate, decreasing= TRUE)
     75758       4401       9202      35542      23108      11743      73436       8578      16194
46.1810604 11.1791318  9.1115509  8.8253954  7.7884602  6.2365402  6.2317441  5.4440845  5.3218069
     68481      13049      57106      27799      73885      17171      74164      27943      75401
 4.7034878  4.5878646  4.4425281  4.1808609  4.1093888  3.9893444  3.3862676  3.2651643  3.0399338
     62995       9043      20220      25493        559        574       7260      27482      17724
 2.9185347  2.5395832  2.4755019  2.4135444  2.3610510  2.1280573  1.8931993  1.8839197  1.7477125
     46010      34409       1480      16861      25488      32131      48820      62384       5033
 1.6054562  1.5203153  1.4459459  1.4264942  1.4182319  1.4133382  1.2094483  1.1865054  1.1860831
     23165      58188       8287      15659       8220      11776      13776      48664      57484
 1.1049050  1.0917800  1.0460685  1.0122406  1.0000000  1.0000000  1.0000000  1.0000000  1.0000000
     51764      75272      22277      41847      43253      41874      11932      10819      21151
 1.0000000  1.0000000  1.0000000  1.0000000  1.0000000  1.0000000  1.0000000  1.0000000  1.0000000
```

We can see that the **highest alpha centrality is 46.1810604 for the node 75758**, implying that this node has a strong influence in the network.
Following this node, the nodes **4401**, **9202**, and **35542** have the alpha centrality **11.1791318**, **9.1115509**, and **8.8253954** respectively. We can see the other alpha centralities in descending order above.

b) Central Node

We can find the central node in two ways, either based on the sum of in and out degree or based on the node with the highest betweenness.

Based on the sum of in and out degree, we found the **central node 8886** as shown below-

```
> most_central <- which.max(degree(g, mode="all"))
> most_central
8886
 156
```

Based on the highest betweenness value, we found the **central node to be the same 8886** as shown below-

```
> bet <- betweenness(g)
> most_central <- which.max(bet)
> most_central
8886
 156
```

c) Longest Path(s)

```
> sg = induced_subgraph(g, which(components (g) $membership == 1))
> V(sg)$degree = degree(sg)
> result = dfs(sg, root = 1, dist = TRUE)$dist
> sort(result, decreasing= TRUE)
61710 67561 36935 61709 61711 63401 63893 38748 69949 57193 61679 61680 61681 61682 41214 41212 41213
 9955  9955  9954  9954  9954  9954  9954  9953  9953  9953  9953  9953  9953  9953  9953  9953  9953
32077 75287 24951 55548 57192 59268 59269 59582 59583 60676 60677 68358 68359 61180 61687 61688 67355
 9952  9952  9952  9952  9952  9952  9952  9952  9952  9952  9952  9952  9952  9952  9952  9952  9952
 4166  9044 32087 49257 24360 30164 55547 32139 39283 57788 57789 28171 59581 61099 28673 62459 67773
 9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9950
30931 13882 25170 46825 32078 32079 25244 15469 54098 18327 55830 55831  6962 20397 34901 61686 66399
```

Since the **longest path** must be in the largest connected component, we extract it using the **components()** function, and create a subgraph containing vertices within that largest connected component. We also assign the degree to each vertex attribute.
We can then use **DFS** to compute the largest distance from one vertex.

Our result above shows that **the longest path is 9955 from nodes 61710 and 67561**.

d) Largest Clique(s)

```
> largest_cliques(g)
[[1]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 26109 45553 75103 38109 226   44441 68882 15553 337   22220 49998 56661 2111  2556  74770 31442
[17] 2222  1111  50109 38775 75547 17775 21108

[[2]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 26109 45553 75103 38109 226   44441 68882 15553 337   22220 14776 56661 75436 1889  2556  17775
[17] 38775 75547 21108 50109 59883 74770 62661

[[3]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 26109 45553 75103 38109 226   44441 68882 15553 337   22220 14776 56661 75436 1889  2556  17775
[17] 38775 75547 21108 50109 59883 74770 2222

[[4]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 26109 45553 33443 68882 38109 226   44441 2222  337   22220 15553 56661 2111  31442 1111  17775
[17] 49998 50109 21108 74770 75547 2556  38775

[[5]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 26109 45553 33443 68882 38109 226   44441 2222  337   22220 15553 56661 1889  14776 75436 38775
[17] 21108 17775 50109 2556  75547 74770 59883

[[6]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 26109 45553 33443 68882 38109 27775 74770 2556  44441 2222  31442 50109 337   22220 15553 17775
[17] 56661 2111  21108 38775 1111  75547 49998

[[7]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 45553 75103 38109 68882 226   44441 448   15553 337   22220 14776 1889  75436 7221  2222
[17] 44442 69993 19997 62550 1778  33554 49997

[[8]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 45553 75103 38109 68882 226   44441 56661 15553 337   33554 2222  44442 22220 62550 49997
[17] 14776 1889  7221  1778  69993 19997 75436

[[9]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 45553 33443 68882 38109 337   2222  226   448   22220 14776 44441 15553 1889  7221  62550
[17] 75436 69993 19997 44442 1778  33554 49997

[[10]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 45553 33443 68882 38109 337   2222  226   56661 15553 44441 33554 22220 14776 1889  42219
[17] 62550 7221  75436 69993 49997 44442 1778

[[11]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 45553 33443 68882 38109 337   2222  226   56661 15553 44441 33554 22220 14776 1889  42219
[17] 62550 7221  75436 69993 49997 44442 17775

[[12]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 45553 33443 68882 38109 337   2222  226   56661 15553 44441 33554 22220 14776 1889  19997
[17] 1778  62550 69993 7221  75436 49997 44442

[[13]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 44441 75103 226   38109 18886 68882 448   2222  1889  337   22220 7221  15553 44442 14776
[17] 49997 62550 33554 1778  19997 69993 75436

[[14]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 44441 75103 226   38109 18886 68882 56661 337   15553 33554 2222  75436 1889  14776 22220
[17] 62550 49997 44442 69993 7221  1778  19997

[[15]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 44441 33443 18886 38109 226   68882 337   2222  22220 1889  448   75436 14776 7221  62550
[17] 69993 19997 15553 1778  33554 44442 49997

[[16]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 44441 33443 18886 38109 226   68882 337   2222  22220 1889  56661 15553 69993 75436 14776
[17] 33554 62550 7221  49997 44442 17775 42219

[[17]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 44441 33443 18886 38109 226   68882 337   2222  22220 1889  56661 15553 69993 75436 14776
[17] 33554 62550 7221  49997 44442 1778  42219

[[18]]
+ 23/75879 vertices, named, from 27e268e:
 [1] 71104 44441 33443 18886 38109 226   68882 337   2222  22220 1889  56661 15553 69993 75436 14776
[17] 33554 62550 7221  49997 44442 1778  19997
```

Here, we use the **largest_cliques()** function to directly find the largest cliques in our graph g as well as the minimum degree required for all vertices in the largest clique, which is 23. It is displayed in the results above.
We can also use the splitting graph and binary search method to get the largest cliques.

e) Ego(s)

We calculated the **egos** for all the vertices as shown. In our result, most of the values get omitted.

```
ego.graph <- igraph::ego(g)
ego.graph
```

```
[[907]]
+ 12/75879 vertices, named, from 65b41cd:
 [1] 24665 8886  20742 26664 59661 326   17832 28553 2     41509 3523  74888

[[908]]
+ 57/75879 vertices, named, from 65b41cd:
 [1] 24776 2556  4222  14998 37775 58772 60550 63327 73281 74780 2745  11621
[13] 12277 24331 30442 32220 33665 36220 38553 47431 47998 48331 2     32232
[25] 68204 6744  14232 1434  1745  8376  66416 73392 75403 10387 74898 61328
[37] 75059 69549 27642 24232 73603 20531 40032 20520 1756  74895 74889 44588
[49] 74896 74891 21487 74897 29387 74890 74892 74894 74899

[[909]]
+ 15/75879 vertices, named, from 65b41cd:
 [1] 24810 559   32219 48565 74130 1923  37776 38331 2     53454 68220 21031
[13] 24076 7144  22568

[[910]]
+ 50/75879 vertices, named, from 65b41cd:
 [1] 24887 371   2222  5000  6554  6999  7665  10553 13331 23331 24732 34442
[13] 36664 73992 659   2224  23665 26665 30997 38331 38553 41442 41997 53442
[25] 62550 2     2268  71993 37954 70993 74903 51775 17876 493   70350 74900
[37] 16054 26065 52553 30409 74901 14732 3790  14721 49487 74905 74902 30287
[49] 74906 74907

[[911]]
+ 63/75879 vertices, named, from 65b41cd:
 [1] 24998 3889  4000  5289  5778  8886  10553 18886 19886 19997 31108 31443
[13] 37775 42197 54441 54442 56661 59994 73281 1923  12266 23554 26109 26665
[25] 28998 30664 31997 32775 33443 34220 34331 34554 43553 64883 69438 2
[37] 2479  67995 75647 15187 20687 70771 32232 75603 7244  15821 3490  2756
[49] 73682 40964 12732 6733  16976 31631 73581 11854 31565 19953 15865 74909
[61] 56284 74908 74910

[[912]]
+ 145/75879 vertices, named, from 65b41cd:
  [1] 25109 4     115   182   1445  4556  5999  6110  6443  6554  7221  7554
 [13] 7665  7998  8664  9442  9997  10220 10331 10775 11443 12443 13331 14887
+ ... omitted several vertices

[[997]]
+ 53/75879 vertices, named, from 65b41cd:
 [1] 32331 4222  5000  8886  13554 19886 20886 23331 33330 33776 44441 63327
[13] 73658 326   24220 30442 34554 37998 45331 2     75647 55552 4457  93
[25] 70993 1734  66549 16043 1434  73392 73614 75571 1401  74118 48431 24265
[37] 66516 9887  75059 60    24466 20809 28498 72771 1512  73213 74314 73642
[49] 28409 75634 74355 40243 75862

[[998]]
+ 35/75879 vertices, named, from 65b41cd:
 [1] 32442 7800  8886  12187 12788 15331 15921 20220 58883 16265 16332 25442
[13] 70549 2     2112  35409 47365 25398 11743 35132 12444 74636 9953  7599
[25] 31409 36221 43942 62806 75176 74615 54620 75864 26987 45599 75863

[[999]]
+ 978/75879 vertices, named, from 65b41cd:
  [1] 32487 1334  2778  2889  3111  3145  3334  4778  5778  6855  7665  12388
 [13] 13065 14276 19997 21109 21553 22332 31664 32576 36553 37775 42031 43330
 [25] 47876 48886 49886 54441 59994 61106 67051 67106 73281 74103 83    326
 [37] 354   939   1337  1375  1589  1750  2301  4211  4531  4801  4890  5001
 [49] 5056  5145  5278  5655  5678  6889  6955  9431  10132 10965 12987 18376
 [61] 18932 24154 24443 27331 27799 31920 35554 36887 36998 39409 44197 44330
 [73] 47121 49553 54498 55264 55331 57473 70039 70549 70882 71305 72550 73249
 [85] 74212 74443 74747 74867 74925 2     2056  2112  74870 3501  9054  15209
 [97] 17854 65883 75581 2232  11888 14643 55020 61784 74970 75847 1952  28032
[109] 29498 75479 7954  37954 1689  33943 35221 75138 2013  73485 16043 60628
+ ... omitted several vertices

[[1000]]
+ 24/75879 vertices, named, from 65b41cd:
 [1] 32553 448   1889  2222  5000  7221  9442  13331 26664 44442 60550 2
[13] 6621  238   3845  73270 3923  18132 3335  3579  44364 19631 31042 75865

[ reached getOption("max.print") -- omitted 74879 entries ]
```

f) Power Centrality

We execute the **power_centrality()** function on the subgraph (as it is not as computationally expensive as running on the original graph) and get following result:

```
> pc <- power_centrality(sg_duplicate, exponent = 0.8)
> sort(pc, decreasing = TRUE)
    27110     74605     32164     74450      6196      9565     11764      3178
3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762
    51587     75528      8947     31527      5181     13891     11774     44941
3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762
    44947      9214       729     47103     48472     12848     36174     13720
3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762
    50127     15939     53795     60549     27842     62785     65065     65572
3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762 3.2779762
    35899     74479     67807     75717     13109     19454     55553      7510
0.1242862 0.1242862 0.1242862 0.1242862 0.0000000 0.0000000 0.0000000 0.0000000
     8145     12198     25998     33710     47887     61295     74826      2085
```

6. Discussion

The project helps us understand how to work with datasets and generating graphs from them, especially medium to large scale datasets. We got hands experience in working with R, the igraph and sna packages, writing our own functions, and more. We gained skills related to plotting and visualizing the graphs, simplifying them, and adjusting the parameters to make a graph readable. Through the various graph analytics functions covered in this project, we gain valuable insights into the structure and relationships within the graph.

We learned about crucial measures in understanding the problem space such as alpha centrality, central node in the graph, longest paths, largest cliques, egos, and power centrality. We also learned about the process of finding communities in a graph through random walks. By understanding these measures, we can identify patterns, structures, and relationships in the data, and develop effective strategies for analyzing and visualizing large and complex networks.

To summarize, after the completion of this project, we are confident in working with R, performing graph analytics on large datasets, generating graphs, plotting them effectively, applying different functions and measures/metrics on the graph that lets us draw different inferences about the graph and the problem domain, along with simplifying graphs for better analysis.